Securing Security Tools

SuriCon 2016

Pierre Chifflier

pierre.chifflier@ssi.gouv.fr

French National Information Security Agency

2016

ANSSI

- Created on July 7th 2009, the ANSSI (French Network and Information Security Agency) is the national authority for the defense and the security of information systems.
- Under the authority of the Prime Minister
- Main missions are:
  - prevention
  - defense of information systems
  - awareness-rising

http://www.ssi.gouv.fr/en/

Securing Security Tools

Objectives of this talk:

- Improving security of tools
- Not on small steps, but trying to solve problems
- Consider alternatives to common solutions
- Test our claims

What is a network IDS ?

A device that

- monitors network for malicious activity
- does stateful protocol analysis
- raises alerts to the administrators
- has to be fast

What is a network IDS ?

From the security point of view, a NIDS is:

- exposed to malicious trafic
- running lots of protocols dissectors
- connected to the admin network
- coded for performance

Root causes

- Bad specifications
  - when they exist
- Design complexity and attack surface
- Formats complexity
- Programming language
- Paradox: many security tools are not securely coded
  - “I’ll fix it later”
  - Infosec people considering it’s “not their job”

Mimimal solutions

- Finding vulns does not (really) help security!
  - But it helps (raising awareness, demonstrating the problem, etc.)
  - The bug is fixed
  - But what about the (probably many) others?
- Fuzzing is not the solution either
  - Level 0 of security audit
  - But it works
- Building secure tools provides much more value
  - It's also much more complicated

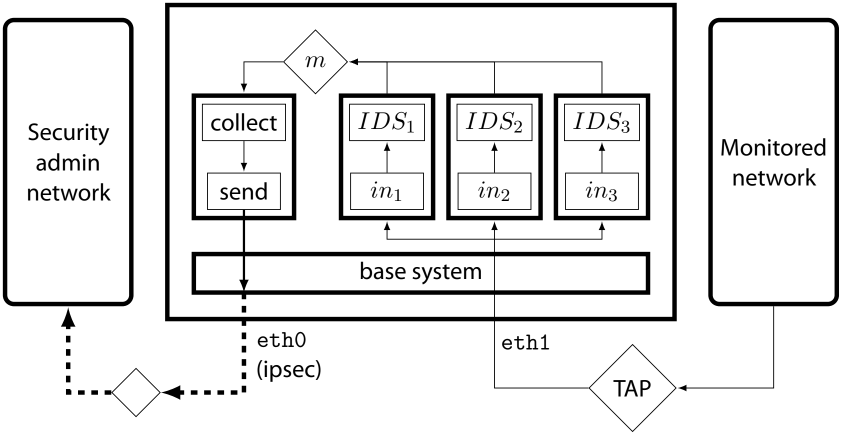
Solutions

- Software environment: minimize consequences of a problem
- Software: try to avoid problems

Architecture Hardening: overview

- Reduced capabilities
- Isolated components
- Write $\oplus$ Execute
- Send-only mechanism for logs
  - Tip: you can write data to a Unix socket in a RO-mounted partition
- Harden kernel
- Read-only containers (everything except `/run`)
- See [CF14] (french)

Architecture



Hardening software

- Reduce attack surface
- Secure design: simple, isolated components
- Managed memory

Suricata

Note on Suricata

- Good points:
  - Security awareness
  - Coding style
  - QA tools: unit tests, build bot, etc.
- But can we get rid of potential memory problems ?
  - Buffer overflows
  - Pointer arithmetic
  - Use-after-free
  - …

Hardening software

Design changes:

- Split components
- Use adequate language

- Easy to say
- Let's try!

The Rusticata proof of concept

Motivations

- Isolate critical code (parsing)
  - Parsers should focus on protocols, not pointers
- Keep performance
- Build robust tools by design

Why not C?

How to code a secure parser in C:

a. defensive programming → fail
b. use QA tools: unit tests, etc. → fail
c. use fuzzing → fail
d. you're the C god! → doubtful

Results: not so good

- Parsing is hard (ex: JSON [Ser16])
- For ex: Wireshark, 60 vulns in 2105, 57 in 2016
- Of course, my own code

Alternatives

- OCaml, Haskell
- Python, Ruby, Perl
- Go, Rust
- C++, Java
- Lua
- Javascript

See [JO14] for why to exclude many of them

Language choice

Yet another language? We want the following properties:

- Easy to embed
- Memory safety
- Strong typing[1]
- Thread safety
- No garbage collector (world stop)
- Fast data exchange with C
- Efficient, avoid useless copies
- Good community

Good candidate: Rust

[1]Which has nothing to do with pressing the keys harder

Overview

Rusticata: 3 main parts

- Suricata: fake app-layer (C)
- Rusticata: glue layer, wraps the C arguments for Rust (Rust)
- Rust parsers: independant projects (Rust)

Notes

- Existing signature engine is used
- Log helper functions too

Parser generator

Nom [G.15] allows to describe data, and generate the parser

Reading bytes:

```
b1 = read_next_byte(&i);
type = b1 as u;
b2 = read_next_byte(&i);
b3 = read_next_byte(&i);
length = b2 >> 8 + b3; // big-endian
```

Describing data:

```
parse_record(&i) {
  type:   be_u8,
  length: be_u16,
}
```

Better readability ⇒ less bugs

Example: the SSL/TLS parser

- Secure almost all internet communications
- Complex protocol [BBDL+15]
- State-oriented parsing
- Multiple layers, application-level fragmentation
- Good comparison with the existing parser[2]

[2] I plead guilty for writing the previous one …

Example of changes (real code)

```c
uint16_t cipher_suites_length =
        input[0] << 8 | input[1];
input += 2;

input += cipher_suites_length;

if (!(HAS_SPACE(1)))
    goto invalid_length;

/* skip compression methods */
uint8_t compression_methods_length =
        *(input++);

input += compression_methods_length;
```

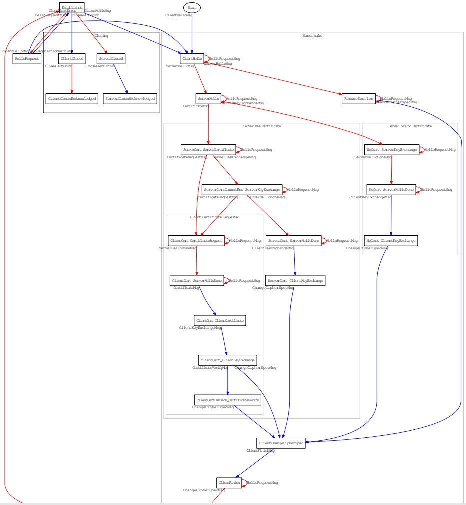
```rust
ciphers_len: be_u16 ~
ciphers: flat_map!(take!(ciphers_len),parse_cipher_suites) ~
comp_len: take!(1) ~
comp: count!(be_u8, comp_len[0] as usize) ~
```

The TLS parser

Skipping to the results (tech. details in other slides)

- covers SSLv3 to TLS 1.2
- more features than the C parser (extensions, defragmentation)
- some parts missing (detection keywords)
- less code: ~400 lines vs 800 for the same features
- rust parser is now ~900 lines
- less time to code
- almost entirely zero-copy
- **no unsafe code**

Bonus: TLS state machine



- New parser offers possibilities to go further
- We can now express more complex security checks
- Extension: represent the TLS state machine
- Detect invalid transitions

Bonus: TLS state machine

Rust representation:

```
match (state,msg) {
  (TlsState::None,         &TlsMessageHandshake::ClientHello(ref msg)) => {
    match msg.session_id {
        Some(_) => Ok(TlsState::AskResumeSession),
        _       => Ok(TlsState::ClientHello)
    }
  },
  // Server certificate
  (TlsState::ClientHello, &ServerHello(_))       => Ok(TlsState::ServerHello),
  (TlsState::ServerHello, &Certificate(_))       => Ok(TlsState::Certificate),
  // Server certificate, no client certificate requested
  (TlsState::Certificate, &ServerKeyExchange(_)) => Ok(TlsState::ServerKeyExchang
```

Match possible on either message type or content

Are we safe now?

Is the problem solved for good?

- Buffer overflows, pointer errors, double frees -> no more!
- Programming logic / algorithmic errors -> still here
- Compiler errors -> can happen

Lessons learned

- Choosing a good language helps
  - Strong typing is great
  - Exhaustive pattern matching
- Cost: learning a new language
  - Lifetimes can be hard (for good reasons)
- Development time: same as C on first parsers, faster after
- Debugging time: **greatly reduced, no debugger required!**
- **No more segfault**

Get the code

- Project main address: `https://github.com/rusticata`
- Suricata fake app-layer + detection
- Rusticata: wraps parsers (only TLS for now)
- Design document in the Rusticata wiki
- Rust parsers:
  - TLS
  - DER
  - NTP
  - SNMP
  - soon: X.509, IKEv2, ...

Conclusion

- Looking at things differently is important
- Try to fix bugs for good
- Memory-safe parsers are a huge security improvement
  - Proof of concept: success
  - Not meant to replace all existing parsers
  - Requires some work to go further
- No global rewrite required, only sensitive code

Questions ?

References

References

[BBDL+15] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, et al.
A messy state of the union: taming the composite state machines of TLS.
In IEEE Symposium on Security & Privacy 2015 (Oakland'15), 2015.

[CF14] P. Chifflier and A. Fontaine.
Architecture système d'une sonde durcie.
Conference C&ESAR, 2014.

[G.15] Couprie G.
Nom, a byte oriented, streaming, zero copy, parser combinators library in rust.
2015 IEEE Security and Privacy Workshops (SPW), 2015.

[JO14] E. Jaeger and Levillain O.
Mind your language(s): A discussion about languages and security.
2014 IEEE Security and Privacy Workshops (SPW), 2014.

[Ser16] N. Seriot.
Parsing json is a minefield.
http://seriot.ch/parsing_json.html, 2016.